CYPR-CD01214

UNITED STATES PATENT APPLICATION FOR

COMBINED IN-CIRCUIT EMULATOR AND PROGRAMMER

Inventors:

Craig Nemecek

Steve Roe

Prepared by:
WAGNER, MURABITO & HAO, LLP
Two North Market Street
Third Floor
San Jose, California 95113
(408) 938-9060

COMBINED IN-CIRCUIT EMULATOR AND PROGRAMMER

CROSS REFERENCE TO RELATED DOCUMENTS

This application is a continuation-in-part of U.S. Patent Application Serial No. 09/975,105 filed October 10, 2001 to Nemecek entitled " Host to FPGA Interface in an In-Circuit Emulation System", which is hereby incorporated. The application is related to, incorporates by reference and claims priority benefit under 35 U.S.C. §119(e) of provisional patent application serial no. 60/243,708 filed October 26, 2000 to Snyder, et al. entitled "Advanced Programmable Microcontroller Device" which is also hereby incorporated herein by reference.

COPYRIGHT NOTICE

FIELD OF THE INVENTION

This invention relates generally to the field of In-Circuit Emulation. More particularly, this invention relates to use of a pod device in an In-Circuit Emulation system to provide both programming and In-Circuit Emulation functions.

BACKGROUND OF THE INVENTION

In-circuit emulation (ICE) has been used by software and hardware developers for a number of years as a development tool to emulate the operation of complex circuit building blocks and permit diagnosis and debugging of hardware and software. Such in-circuit emulation is most commonly used currently to analyze and debug the behavior of complex devices such as microcontrollers and microprocessors that have internal structures that are far too complex to readily model using computer simulation software alone.

FIGURE 1 illustrates an exemplary conventional in-circuit emulation arrangement 100 used to model, analyze and debug the operation of a microcontroller device. In this arrangement, a host computer (e.g., a personal computer) 110 is connected to a debug logic block 120 which is further connected to a special version of the microcontroller device that has been developed specially for use in emulation. Operational instructions are loaded from the host computer 110 through the debug logic 120 to the special version of the microcontroller 130. The debug logic 120 monitors operation of the microcontroller 130 as the instructions are executed. Depending upon the application, this operation may be monitored while the special version of the microcontroller 130 is interconnected with the circuitry that is intended to interface a production version of the microcontroller in the finished product under development. Such interconnection may be via simulation within host computer 110 or as actual circuitry or some combination thereof. As the circuit is stepped through its operation, the debug logic gathers information about the state of various components of the microcontroller 130 during operation and feeds that information back to the host computer 110 for analysis.

During the course of the analysis, various trace information such as time stamps, register values, data memory content, etc. may be logged in the host computer 110 for analysis and debugging by the designer. Additionally, it is generally the case that various break points can be defined by the designer that

cause the program to halt execution at various points in the operation to permit detailed analysis. Other debugging tools may also be provided to enable the user to debug the operation of the circuit.

In-circuit emulation systems such as 100 have a number of disadvantages and limitations. In earlier systems, the microcontroller 130 might have been simply the production version of the microcontroller itself with no special debugging features. In such systems, the information that can be gathered by the ICE system 100 is limited to that which is available at the pinouts of the microcontroller 130 (or which can be extracted from the microcontroller using clever programming or special coding supported by the processor).

Enhancements to such early systems provided the microcontroller or other processor with an array of built-in debugging tools that use standard pins on the part and built-in instructions that facilitated in-circuit emulation. In such enhanced processors, the emulation tools are integrated into the part and thus become a design constraint for developing and improving the part. Thus, support for the debugging instruction code and the like can increase the cost and complexity of the circuit.

Newer systems often use a so-called "bond-out" microcontroller. A bond-out version of the microcontroller is a version of the production microcontroller that has been designed with special wirebonding pads on the chip that are not normally connected in the production wirebonding. The bond-out version connects these pads to pins on the microcontroller package to permit access to otherwise inaccessible points of the circuit to facilitate debugging. This technique is in common use, but has the disadvantage of imposing significant limitations on the circuit layout to permit space and circuitry associated with the special wirebonding pads. Additionally, it is usually the case that substantial interface circuitry and other special circuitry to facilitate the debugging and bond-out has to be added to the circuit. This increases the complexity, size, power consumption and potentially reduces the yield of the production part. Moreover, development resources are required to lay out the bond-out circuitry and pads and do associated design of

such bond-out circuitry. Additionally, instruction code must generally be provided and supported for such an implementation. Such resources may have to be applied with every updated version of the part and may significantly impact speed, cost or flexibility in development of improved versions of the part.

A third technique, one that is used in the Pentium™ and Pentium Pro™ series of microprocessors available from Intel Corporation, provides a special "probe mode" of operation of the processor. When the processor is placed in this probe mode, a number of internal signals are routed to a "debug port" for use by the in-circuit emulation system. This debug port is used to permit the in-circuit emulation system to communicate with the processors at all times and, when placed in probe mode, to read otherwise inaccessible probe points within the processor. Of course, providing such a probe mode requires significant design resources to design in all such probe and debug functions and associated instruction code support into the standard processor. This, of course, increases development cost, chip complexity and chip size. Moreover, such facilities become a part of the processor design which must be carried through and updated as required as enhancements to the original design are developed.

SUMMARY OF THE INVENTION

The present invention relates generally to providing device programming capabilities in an ICE system. Objects, advantages and features of the invention will become apparent to those skilled in the art upon consideration of the following detailed description of the invention.

In one embodiment consistent with the present invention, a combined In-Circuit Emulation system and device programmer has a pod assembly used in an in-circuit emulation system has both a real microcontroller used in the In-Circuit Emulation and debugging process as well as a socket that accommodates a microcontroller to be programmed (a program microcontroller). Programming can be carried out over a single interface that is shared between the microcontroller and the program microcontroller and which is also used to provide communication

between the real microcontroller and the In-Circuit Emulation system to carry out emulation functions. In order to assure that the emulation microcontroller does not interfere with the programming process for a microcontroller placed in a programming socket, a special sleep mode is implemented in the emulation microcontroller. This sleep mode is activated by a process that takes place at power on in which the a reset line is released with a specified data line held in a logic high state. This provides the advantage of using a single pod and ICE system to also provide programming functions without need for a separate programmer.

A combined in-circuit emulation system and programmer consistent with certain embodiments of the present invention has a pod carrying an emulation microcontroller and a socket for programming another microcontroller. A base station has a virtual microcontroller that operates in lock-step synchronization with the emulation microcontroller during emulation operations. An interface connects the pod to the base station, the interface having a clock signal line, a pair of data signal lines, a reset line and a power line, wherein the reset line is connected to the emulation microcontroller, but is not connected to the socket. The emulation microcontroller can be placed in a sleep mode so that a microcontroller residing in the socket can be programmed by receiving programming information from the base station without the programming being disturbed by actions of the emulation microcontroller.

A pod assembly for use with a combined In-Circuit Emulation system and device programmer consistent with certain embodiments of the present invention has a device under test and a programming socket for carrying a device to be programmed. An interface connects data lines to programming inputs of the device to be programmed and to the device under test. The data lines carry programming instructions to the device to be programmed during a programming operation, and the data lines are used for communication with the device under test during emulation operations.

A method of programming a programmable device consistent with certain embodiments of the present invention, wherein the device residing in a socket of

a pod that carries an emulation device used in emulation operations, includes applying power to the pod; sending a control signal to the pod to place the emulation device into a sleeping state; and programming the programmable device residing in the socket while the emulation device is in the sleeping state.

The above summaries are intended to illustrate exemplary embodiments of the invention, which will be best understood in conjunction with the detailed description to follow, and are not intended to limit the scope of the appended claims.

BRIEF DESCRIPTION OF THE DRAWINGS

The features of the invention believed to be novel are set forth with particularity in the appended claims. The invention itself however, both as to organization and method of operation, together with objects and advantages thereof, may be best understood by reference to the following detailed description of the invention, which describes certain exemplary embodiments of the invention, taken in conjunction with the accompanying drawings in which:

FIGURE 1 is a block diagram of a conventional In-Circuit Emulation system.

FIGURE 2 is a block diagram of an exemplary In-Circuit Emulation system consistent with certain microcontroller embodiments of the present invention.

FIGURE 3 is an illustration of the operational phases of an In-Circuit Emulation system consistent with an embodiment of the present invention.

FIGURE 4 is an illustration of the operational phases of an In-Circuit Emulation system consistent with an embodiment of the present invention viewed from a virtual microcontroller perspective.

FIGURE 5 is a timing diagram useful in understanding an exemplary data and control phase of operation of certain embodiments of the present invention.

FIGURE 6 is a block diagram isolating the host to FPGA interface consistent with an embodiment of the present invention

FIGURE 7 is a flow chart describing the operation of the host to FPGA interface in an embodiment consistent with the present invention.

FIGURE 8 is a block diagram of a pod assembly consistent with certain embodiments of the present invention.

FIGURE 9 is a flow chart of a process for entering a debug mode consistent with an embodiment of the present invention.

FIGURE 10 is a flow chart of a process for entering a programming mode consistent with an embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

In the following detailed description of the present invention, numerous specific details are set forth in order to provide a thorough understanding of the present invention. However, it will be recognized by one skilled in the art that the present invention may be practiced without these specific details or with equivalents thereof. In other instances, well known methods, procedures, components, and circuits have not been described in detail as not to unnecessarily obscure aspects of the present invention.

NOTATION AND NOMENCLATURE

Some portions of the detailed descriptions which follow are presented in terms of procedures, steps, logic blocks, processing, and other symbolic representations of operations on data bits that can be performed on computer memory. These descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. A procedure, computer executed step, logic block, process, etc., is here, and generally, conceived to be a self-consistent sequence of steps or instructions leading to a desired result. The steps are those requiring physical manipulations of physical quantities.

Usually, though not necessarily, these quantities take the form of electrical

or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated in a computer system. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the following discussions, it is appreciated that throughout the present invention, discussions utilizing terms such as "processing" or "transferring" or "executing" or "releasing" or "instructing" or "issuing" or "halting" or "clearing" or the like, refer to the action and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

COMBINED IN-CIRCUIT EMULATOR AND PROGRAMMER IN ACCORDANCE WITH THE INVENTION

While this invention is susceptible of embodiment in many different forms, there is shown in the drawings and will herein be described in detail specific embodiments, with the understanding that the present disclosure is to be considered as an example of the principles of the invention and not intended to limit the invention to the specific embodiments shown and described. In the description below, like reference numerals are used to describe the same, similar or corresponding parts in the several views of the drawings.

A commercial ICE system utilizing the present invention is available from Cypress Micro Systems, Inc., for the CY8C25xxx/26xxx series of microcontrollers. Detailed information regarding this commercial product is available from Cypress

Micro Systems, Inc., 22027 17th Avenue SE, Suite 201, Bothell, WA 98021Bothell, WA in the form of version 1.11 of "PSoC Designer: Integrated Development Environment User Guide", which is hereby incorporated by reference. While the present invention is described in terms of an ICE system for the above exemplary microcontroller device, the invention is equally applicable to other complex circuitry including microprocessors and other circuitry that is suitable for analysis and debugging using in-circuit emulation. Moreover, the invention is not limited to the exact implementation details of the exemplary embodiment used herein for illustrative purposes.

Referring now to **FIGURE 2**, an architecture for implementation of an embodiment of an ICE system of the present invention is illustrated as system 200. In system 200, a Host computer 210 (e.g., a personal computer based on a Pentium™ class microprocessor) is interconnected (e.g., using a standard PC interface 214 such as a parallel printer port connection, a universal serial port (USB) connection, etc.) with a base station 218. The host computer 210 generally operates to run an ICE computer program to control the emulation process and further operates in the capacity of a logic analyzer to permit a user to view information provided from the base station 218 for use in analyzing and debugging a system under test or development.

The base station 218 is based upon a general purpose programmable hardware device such as a gate array configured to function as a functionally equivalent "virtual microcontroller" 220 (or other device under test (DUT)). This is accomplished using an associated integral memory 222 which stores program instructions, data, trace information and other associated information. Thus, the base station is configured as an emulator of the internal microprocessor portion of the microcontroller 232. In preferred embodiments, a field programmable gate array FPGA (or other programmable logic device) is configured to function as the virtual microcontroller 220. The FPGA and virtual microcontroller 220 will be referred to interchangeably herein. The base station 218 is coupled (e.g., using a four wire interface 226) to a standard production microcontroller 232 mounted in a

mounting device referred to as a "pod". The pod, in certain embodiments, provides connections to the microcontroller 232 that permit external probing as well as interconnection with other circuitry as might be used to simulate a system under development.

The FPGA of the base station 218 of the current embodiment is designed to emulate the core processor functionality (microprocessor functions, Arithmetic Logic Unit functions and RAM and ROM memory functions) of the Cypress CY8C25xxx/26xxx series microcontrollers. The CY8C25xxx/26xxx series of microcontrollers also incorporates I/O functions and an interrupt controller as well as programmable digital and analog circuitry. This circuitry need not be modeled using the FPGA 220. Instead, the I/O read information, interrupt vectors and other information can be passed to the FPGA 220 from the microcontroller 232 over the interface 226 as will be described later.

In order to minimize the need for any special ICE related functions on the microcontroller 232 itself, the FPGA 220 and associated circuitry of the base station 218 are designed to operate functionally in a manner identically to that of microprocessor portion of the production microcontroller, but to provide for access to extensive debug tools including readout of registers and memory locations to facilitate traces and other debugging operations.

The base station 218's virtual microcontroller 220 operates to execute the code programmed into the microcontroller 232 in lock-step operation with the microcontroller 232. Thus, the actual microcontroller 232 is freed of any need to provide significant special facilities for ICE, since any such facilities can be provided in the virtual microcontroller 220. The base station 218's virtual microcontroller 220 and microcontroller 232 operate together such that I/O reads and interrupts are fully supported in real time. The combination of real and virtual microcontroller behave just as the microcontroller 232 would alone under normal operating conditions. I/O reads and interrupt vectors are transferred from the microcontroller 232 to the base station 218 as will be described later. Base station 218 is then able to provide the host computer 210 with the I/O reads and interrupt

vectors as well as an array of information internal to the microcontroller 232 within memory and register locations that are otherwise inaccessible.

In the designing of a microcontroller other complex circuit such as the microcontroller 232, it is common to implement the design using the Verilog™ language (or other suitable language). Thus, it is common that the full functional design description of the microcontroller is fully available in a software format. The base station 218 of the current embodiment is based upon the commercially available Spartan™ series of FPGAs from Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. The Verilog™ description can be used as the input to the FPGA design and synthesis tools available from the FPGA manufacturer to realize the virtual microcontroller 220 (generally after timing adjustments and other debugging). Thus, design and realization of the FPGA implementation of an emulator for the microcontroller (virtual microcontroller) or other device can be readily achieved by use of the Verilog™ description along with circuitry to provide interfacing to the base station and the device under test (DUT).

In the embodiment described in connection with **FIGURE 2**, the actual production microcontroller 232 carries out its normal functions in the intended application and passes I/O information and other information needed for debugging to the FPGA 220. The virtual microcontroller 220 implemented within the FPGA of base station 218 serves to provide the operator with visibility into the core processor functions that are inaccessible in the production microcontroller 232. Thus, the FPGA 220, by virtue of operating in lock-step operation with the microcontroller 232 provides an exact duplicate of internal registers, memory contents, interrupt vectors and other useful debug information. Additionally, memory 222 can be used to store information useful in trace operations that is gathered by the FPGA 220 during execution of the program under test. This architecture, therefore, permits the operator to have visibility into the inner workings of the microcontroller 232 without need to provide special bondouts and expensive circuitry on the microcontroller itself.

The base station 218's FPGA based virtual microcontroller 220, operating

under control of host computer 210, carries out the core processor functions of microcontroller 232 and thus contains a functionally exact emulated copy of the contents of the registers and memory of the real microcontroller 232. The ICE system starts both microcontrollers (real and virtual) at the same time and keeps them running in synchronization. The real microcontroller 232 sends I/O data to the base station 218 (and in turn to the ICE software operating on the host computer 210 if required) fast enough to keep the real microcontroller 232 and the virtual microcontroller 220 of base station 218 in synchronization. Whenever the system is halted (i.e., when the system is not emulating), other information such as flash memory programming functions, test functions, etc. can be sent over the interface.

Because the microcontroller 232 operates in synchronization with the virtual microcontroller 220, less data needs to be sent over the four wire interface than would be required in an ICE system otherwise. The type of data sent over the lines is allowed to change depending on when the data is sent in the execution sequence. In other words, depending on the execution sequence time, the information over the data lines can be commands to the real microcontroller 232 or they can be data. Since the clock frequency of the real microcontroller 232 is programmable, it copies its current clock on one of the lines of the four wire interface. Moreover, the lock-step operation of the microcontroller 232 and the virtual microcontroller 220 allows the virtual microcontroller 220 to not require certain resources of the microcontroller 232 such as timers, counters, amplifiers, etc. since they are fully implemented in the microcontroller 232. In addition, the microcontroller 232 (or other DUT) can be debugged in real time without need for extensive debug logic residing on the microcontroller 232, since all registers and memory locations, etc. are available through the virtual microcontroller 220.

In the embodiment illustrated, the basic interface used is a four line interface between microcontroller 232 and base station 218. This interface permits use of a standard five wire Category Five patch cable to connect the microcontroller 232 and base station 218 in one embodiment, but of course, this is not to be considered

limiting. The four wire interface 226 of the present embodiment can be functionally divided into two functional portions. A data transport portion 242 carries two data lines in the current embodiment. A clock portion 246 carries a debug system clock plus the microcontroller clock signal for the microcontroller 232. Three additional lines are also provided (not shown) for supply, ground and a reset line. But, the data transport portion 242 and the clock portion 246 are of primary interest, since the supply and reset functions can be readily provided in any other suitable manner.

The two portions of the interface are implemented in the current embodiment using four lines as described, however, in other embodiments, these two portions can be implemented with as few as two wires. In the current embodiment, the microcontroller clock signal can be varied by programming (even dynamically during execution of a program). Therefore, it is desirable to have two clock signals - the microcontroller clock to easily track the microcontroller clock timing as well as a system clock that regulates the data transfer and other operations. However, in other embodiments, particularly where a clock frequency is not changed dynamically, a single clock can be used. The single clock can be multiplied or divided as required to implement the required clocking signals.

The present embodiment using an eight bit microcontroller that only reads eight bits at a time on any given I/O read. Thus, the present microcontroller 232 needs only to effect serializing and transferring a maximum of one eight bit I/O read for each instruction cycle. This is easily accommodated using two data lines transferring four bits each over four system clock cycles. However, using a clock which is two times faster, a single line could equally well transfer the data in the same time. Similarly, four lines could be used to transfer the same data in only two clock cycles. In any case, the objective is to transfer the data in a short enough time to permit the virtual microcontroller 220 to process the data and issue any needed response before the next instruction cycle begins. The time required to accomplish this is held at a minimum in the current invention, since the system synchronization eliminates need for any overhead protocol for transmission of the data.

The current embodiment of the invention uses a four line communication interface and method of communicating between the FPGA within base station 218 (acting as. a "virtual microcontroller" 220 or ICE) and the real microcontroller device under test (microcontroller 232). The four line communication interface is time-dependent so that different information can be transferred at different times over a small number of communication lines. Moreover, since the two processors operate in lockstep, there is no need to provide bus arbitration, framing, or other protocol overhead to effect the communication between the microcontroller 232 and the virtual microcontroller 220. This interface is used for, among other things, transferring of I/O data from the microcontroller 232 to the FPGA 220 (since the FPGA emulates only the core processor functions of the microcontroller in the current embodiment). A first interface line (Data1) is a data line used by the microcontroller 232 to send I/O data to the FPGA based virtual microcontroller 220. This line is also used to notify the FPGA 220 of pending interrupts. This Data1 line is only driven by the real microcontroller 232. A second data line (Data2), which is bidirectional, is used by the microcontroller 232 to send I/O data to the FPGA based virtual microcontroller of base station 218. In addition, the FPGA 220 uses the Data2 line to convey halt requests (i.e., to implement simple or complex breakpoints) to the microcontroller 232.

A third interface line is a 24/48 Mhz debug system clock used to drive the virtual microcontroller 220's communication state machines (the logic used within the state controller to communicate with the microcontroller 232). In the current embodiment, this clock always runs at 24 MHz unless the microcontroller 232's internal clock is running at 24 Mhz. In this case the system clock switches to 48 Mhz. Of course, these exact clock speeds are not to be considered limiting, but are presented as illustrative of the current exemplary embodiment. The fourth interface line is the internal microcontroller clock from the microcontroller 232.

A fifth line can be used to provide a system reset signal to effect the simultaneous startup of both microcontrollers. This fifth line provides a convenient

mechanism to reset the microcontrollers, but in most environments, the simultaneous startup can also be effected in other ways including switching of power. The reset line can be advantageously used as part of a programming operation as will be described later. Sixth and Seventh lines are provided in the current interface to provide power and ground for power supply.

The base station 218's virtual microcontroller 220 communicates with the microcontroller 232 via four signal and clock lines forming a part of the four line interface 226 forming a part of a seven wire connection as described below. The interface signals travel over a short (e.g., one foot) of CAT5 network cable. The ICE transmits break commands to the microcontroller 232 via the base station 218, along with register read/write commands when the microcontroller 232 is halted. The microcontroller 232 uses the interface to return register information when halted, and to send I/O read, interrupt vector, and watchdog information while running. The microcontroller 232 also sends a copy of its internal clocks for the ICE. The four lines of the four line interface are the first four entries in the table below. Each of the signals and their purpose is tabulated below in **TABLE 1**:

Signal Name	Signal Direction with Respect to Base Station 218	Description
U_HCLK (Data Clock or HCLOCK)	In	24/48MHz data clock driven by microcontroller 232. This clock is used to drive the ICE virtual microcontroller communication state machines. This clock always runs at 24MHz, unless the U_CCLK clock is running at 24MHz — then it switches to 48MHz.
U_CCLK (microcontroller Clock or CCLOCK)	In	The internal microcontroller 232 CPU clock.
U_D1_IRQ (Data1)	In	One of two data lines used by the microcontroller 232 to send I/O data to the ICE. This line is also used to notify the ICE of pending interrupts. This line is only driven by the microcontroller 232 (i.e., unidirectional).
U_D0_BRQ (Data2)	In/Out	One of two data lines used by the microcontroller 232 to send I/O data to the ICE. The ICE uses this line to convey halt requests and other information to the microcontroller 232. This line is used for bi-directional communication.
ICE_POD_RST (RESET)	Out	Optional active high reset signal to microcontroller 232.
ICE_POD_PW_R (POWER)	Out	Optional power supply to microcontroller 232.
ICE_POD_GND (GROUND)	Out	Optional ground wire to microcontroller 232.

TABLE 1

Synchronization between the microcontroller 232 and the virtual microcontroller 220 is achieved by virtue of their virtually identical operation. They are both started simultaneously by a power on or reset signal. They then track

each other's operation continuously executing the same instructions using the same clocking signals. The system clock signal and the microcontroller clock signal are shared between the two microcontrollers (real and virtual) so that even if the microprocessor clock is changed during operation, they remain in lock-step.

In accordance with certain embodiments of the invention, a mechanism is provided for allowing the FPGA 220 of base station 218 and the microcontroller 232 to stop at the same instruction in response to a breakpoint event (a break or halt). The FPGA 220 has the ability monitor the microcontroller states of microcontroller 232 for a breakpoint event, due to its lock-step operation with microcontroller 232. In the process of executing an instruction, an internal start of instruction cycle (SOI) signal is generated (by both microcontrollers) that indicates that the device is about to execute a next instruction. If a breakpoint signal (a halt or break signal - the terms "halt" and "break" are used synonymously herein) is generated by the FPGA, the execution of the microcontroller 232 can be stopped at the SOI signal point before the next instruction starts.

Although the SOI signal is labeled as a signal indicating the start of an instruction, the SOI signal is used for multiple purposes in the present microcontroller. It is not required that the SOI signal actually indicate a start of instruction for many purposes, merely that there be a convenient time reference on which to base certain actions. For example, any reference signal that always takes place prior to execution of an instruction can be used as a time reference for reading a halt command. Accordingly, any such available or generated reference signal can be used equivalently as a "halt read" signal without departing from the present invention. That not withstanding, the SOI signal is conveniently used in the current embodiment and will be used as a basis for the explanation that follows, but should not be considered limiting.

Logic within the FPGA 220 of base station 218 allows not only for implementation of simple breakpoint events, but also for producing breakpoints as a result of very complex events. By way of example, and not limitation, a breakpoint can be programmed to occur when a program counter reaches 0x0030,

an I/O write is happening and the stack pointer is about to overflow. Other such complex breakpoints can readily be programmed to assist in the process of debugging. Complex breakpoints are allowed, in part, also because the virtual microcontroller 220 has time to carry out complex computations and comparisons after receipt of I/O data transfers from the microcontroller 232 and before the next instruction commences. After the receipt of I/O data from the microcontroller 232, the FPGA 220 of base station 218 has a relatively long amount of computation time to determine if a breakpoint event has occurred or not. In the event a breakpoint has occurred, the microcontroller 232 can be halted and the host processor 210 is informed.

An advantage of this process is that the FPGA 220 and the microcontroller 232 can be stopped at the same time in response to a breakpoint event. Another advantage is that complex and robust breakpoint events are allowed while still maintaining breakpoint synchronization between the two devices. These advantages are achieved with minimal specialized debugging logic (to send I/O data over the interface) and without special bond-out circuitry being required in the microcontroller device under test 232.

Normal operation of the current microcontroller is carried out in a cycle of two distinct stages or phases as illustrated in connection with **FIGURE 3**. The cycle begins with the initial startup or reset of both the microcontroller 232 and the virtual microcontroller 220 at 304. Once both microcontrollers are started in synchronism, the data phase 310 is entered in which serialized data is sent from the microcontroller to the virtual microcontroller. At the start of this phase the internal start of instruction (SOI) signal signifies the beginning of this phase will commence with the next low to high transition of the system clock. In the current embodiment, this data phase lasts four system clock cycles, but this is only intended to be exemplary and not limiting. The SOI signal further indicates that any I/O data read on the previous instruction is now latched into a register and can be serialized and transmitted to the virtual microcontroller. Upon the start of the data phase 310, any such I/O read data (eight bits of data in the current embodiment)

is serialized into two four bit nibbles that are transmitted using the Data0 and Data1 lines of the current interface data portion 242. One bit is transmitted per data line at the clock rate of the system clock. Thus, all eight bits are transmitted in the four clock cycles of the data transfer phase.

At the end of the four clock cycle data transfer phase in the current embodiment, the control phase 318 begins. During this control phase, which in the current embodiment may be as short as two microcontroller clock periods (or as long as about fourteen clock periods, depending upon the number of cycles required to execute an instruction), the microcontroller 232 can send interrupt requests, interrupt data, and watchdog requests. Additionally, the virtual microcontroller 220 can issue halt (break) commands. If a halt command is issued, it is read by the microcontroller at the next SOI signal. Once the control phase ends, the data transfer phase repeats. If there is no data to transfer, data1 and data2 remain idle (e.g., at a logic low state). To simplify the circuitry, I/O bus data are sent across the interface on every instruction, even if it is not a bus transfer. Since the virtual microcontroller 220 is operating in synchronization with microcontroller 232 and executing the same instructions, the emulation system knows that data transferred during non I/O read transfers can be ignored.

FIGURE 4 shows this operational cycle from the perspective of the virtual microcontroller 220. During the data transfer phase 310, the serialized data is received over Data0 and Data1. It should be noted that prior to receipt of this I/O data, the microcontroller 232 has already had access to this data for several clock cycles and has already taken action on the data. However, until receipt of the I/O read data during the data transfer phase 310, the virtual microcontroller 220 has not had access to the data. Thus, upon receipt of the I/O read data during the data phase 310, the virtual microcontroller 220 begins processing the data to catch up with the existing state of microcontroller 232. Moreover, once the I/O data has been read, the host computer 210 or virtual microcontroller 220 may determine that a complex or simple breakpoint has been reached and thus need to issue a break request. Thus, the virtual microcontroller should be able to process the data quickly

enough to make such determinations and issue a break request prior to the next SOI. Break requests are read at the internal SOI signal, which also serves as a convenient reference time marker that indicates that I/O data has been read and is available for transmission by the microcontroller 232 to the virtual microcontroller 220.

By operating in the manner described, any breakpoints can be guaranteed to occur in a manner such that both the virtual microcontroller 220 and the microcontroller 232 halt operation in an identical state. Moreover, although the virtual microcontroller 220 and the microcontroller 232 operate on I/O data obtained at different times, both microcontrollers are in complete synchronization by the time each SOI signal occurs. Thus, the virtual microcontroller 220 and the microcontroller 232 can be said to operate in lock-step with respect to a common time reference of the SOI signal as well as with respect to execution of any particular instruction within a set of instructions being executed by both virtual microcontroller 220 and the microcontroller 232.

A transfer of I/O data as just described is illustrated with reference to the timing diagram of **FIGURE 5**. After the microcontroller 232 completes an I/O read instruction, it sends the read data back to the base station 218 to the virtual microcontroller, since the virtual microcontroller 220 of the present embodiment implements only the core processor functions (and not the I/O functions). The ICE system can expect the incoming data stream for an I/O read to commence with the first positive edge of U_HCLK (the debug or system clock) when SOI signal for the following instruction is at a predetermined logic level (e.g., a logic high). Thus, at time T1, the SOI signal makes a transition to a logic high and one system clock cycle later at time T2, the data transfer phase 310 begins. This timing allows the ICE system to get the read data to the emulated accumulator of base station 218 before it is needed by the next instruction's execution. Note that the first SOI pulse shown in **FIGURE 5** represents the first SOI following the I/O read instruction (but could be any suitable reference time signal). Transfer of the data from the

microcontroller 232 is carried out using the two data lines (data2 and data1, shown as U_D0_BRK and U_D1_IRQ) with each line carrying four bits of an eight bit word. During this data transfer phase 310, an eight bit transfer representing the I/O read data can take place from the microcontroller 232 to the base station 210 in the four clock cycles between T2 and T3. The control phase 318 starts at time T3 and continues until the beginning of the next data transfer phase 310. The SOI signal at T4 indicates that the next data transfer phase is about to start and serves as a reference time to read the data2 line to detect the presence of any halt signal from the virtual microcontroller 220. The current control phase 318 ends at T5 and the next data transfer phase 310 begins.

The base station 218 only transmits break (halt) commands to the microcontroller 232 during the control phase. After the microcontroller 232 is halted in response to the break command, the interface can be used to implement memory / register read / write commands. The halt command is read at the SOI signal transition (T1 or T4). The microcontroller 232 uses the interface to return register information when halted, and to send I/O read, interrupt vector and watchdog timer information while running.

To summarize, a break is handled as follows: The ICE asserts U_D0_BRQ (break) to stop the microcontroller 232. When the ICE asserts the break, the microcontroller 232 reads it at the SOI transition to high and stops. The ICE assert breaks during the control phase. The microcontroller 232 samples the U_D0_BRQ line at the rising edge of SOI (at T4) to determine if a break is to take place. After halting, the ICE may issue commands over the U_D0_BRQ line to query the status of various registers and memory locations of the virtual microcontroller or carry out other functions.

In the case of an interrupt, if an interrupt request is pending for the microcontroller 232, the system asserts U_D1_IRQ as an interrupt request during the control phase of the microcontroller 232. Since the interrupt signal comes to the virtual microcontroller 220 from the microcontroller 232 during the control phase, the virtual microcontroller 220 knows the timing of the interrupt signal going

forward. That is, the interrupt signal is the synchronizing event rather than the SOI signal. In case of an interrupt, there is no SOI, because the microcontroller 232 performs special interrupt processing including reading the current interrupt vector from the interrupt controller. Since program instructions are not being executed during the interrupt processing, there is no data / control phase. The virtual microcontroller 220 expects the interrupt vector to be passed at a deterministic time across the interface during this special interrupt processing and before execution of instructions proceeds. Since the virtual microcontroller 220 of the current embodiment does not implement an interrupt controller, interrupt vectors are read from the interrupt controller upon receipt of an interrupt request over the interface. The interrupt vector data is passed over the interface using the two data lines as with the I/O read data, following the assertion of an internal microcontroller IVR_N (active low) signal during the control phase. In the current embodiment, an interrupt cycle is approximately 10 clock cycles long. Since the interrupt service cycle is much longer than the time required to transfer the current interrupt vector, the data is easily transferred using the two data lines, with no particular timing issues.

If the microcontroller 232 undergoes a watchdog reset, it asserts the IRQ (interrupt) and BRQ (break) lines indefinitely. The ICE detects this condition and further detects that the microcontroller clock has stopped. This is enough to establish that a watchdog reset has occurred. The ICE applies an external reset, and notifies the ICE software in the host computer 210.

Referring now to the block diagram of **FIGURE 6**, the interface between the host processor 210 and the base station 218 of a preferred embodiment of the present invention is illustrated. In this embodiment, the connection between the host processor 210 and the FPGA 220 is advantageously provided using a standard IEEE 1284 parallel printer cable 214 with communication carried out using a modification of standard EPP (enhanced parallel port) communication protocol. Of particular interest in this communication interface is the data strobe connection 412, the INIT (initialize) connection 416 and the eight data connections (data line

0 through data line 7) 420. These connection are directly connected to the FPGA with the INIT connection connected to the FPGA RESET pin. The data strobe line 412 is connected to the FPGA configuration clock input and the eight data lines 420 are connected to data input pins of the FPGA.

When the software on the host is started, the INIT connection 416 is driven by the host computer 210 to a logic low causing the FPGA to clear its configuration memory 424 and begin receiving configuration data. The configuration data is stored in configuration memory to define the functionality of the FPGA. This configuration data is clocked in eight bits at a time over the data lines 420 using the data strobe signal as a clock signal. That is, an eight bit word is placed on the interface data lines 420 by host processor 210 followed by toggling the data strobe line to clock the data into the FPGA 220. This unidirectional data transfer from the host computer incorporates a set of design parameters that configure the circuitry of the FPGA 220 to function, in part, as a standard IEEE 1284 EPP interface once the FPGA 220 is programmed and functional. This programming configures the FPGA 220 to have an IEEE 1284 EPP interface with the data lines 420 connected to the FPGA as bidirectional data lines, the configuration clock configured to operate as the IEEE 1284 data clock line connected to data strobe 412 and the INIT line 416 continues to drive the FPGA clear and reset function.

Data transfer continues in this manner until the FPGA 220 is fully programmed by virtue of having received the correct amount of data required by the particular FPGA 220 used in base station 218. Thus, each time the host software is initialized, a data transfer to the FPGA 220 occurs to program the FPGA 220 to function in its capacity of a virtual microcontroller (in this embodiment). Once programming ceases, the FPGA 220 "wakes up" as a virtual microcontroller (or whatever device is programmed into the FPGA 220 in general) and begins to function as the virtual microcontroller. At this point, the interface 214 ceases to function as a unidirectional programming interface and begins to function as a bidirectional communication interface using the programmed operation of the FPGA 220 communicating through its programmed IEEE 1248 EPP parallel

communication interface.

In the virtual microcontroller mode of operation of the FPGA 220, communication is carried out using the eight data lines 420 as bidirectional data lines compliant with IEEE 1284 EPP parallel communication protocol with the data strobe line 412 used as a data clock and the INIT line 416 continuing to act as a clear and reset signal. INIT line 416 can thus be used to reinitialize the programming of the FPGA 220, for example, to revise a design parameter or to simply restart the ICE system. **TABLE 2** below summarizes the significant connections of this interface.

Interface Lines	Program Mode Function	Free Running "Awake" Mode Function
Data bits 0 through 7	Unidirectional data into the FPGA	Bidirectional EPP compliant communication
Data Strobe	Unidirectional programming clock	EPP Compliant Data Strobe
INIT	Low signal indicates clear configuration memory and prepare to receive new configuration data	Low signal indicates clear configuration memory and enter programming mode - prepare to receive new configuration data

TABLE 2

The programming and communication process between the host 210 and the FPGA 220 is described in flow chart 500 of **FIGURE 7** starting at 502. The host software is loaded and initialized at 506, and asserts a logic low on the INIT line 416 to signal a reset and clearing of the FPGA 220's configuration memory 424 at 510. In response to this signal, the FPGA 220 clears configuration memory 424 at 514. The Host computer 210 then begins transferring a new set of configuration parameters to the FPGA 220 at 520 by strobing data into the FPGA's configuration memory 424. This set of configuration parameters configures the FPGA 220 to

have an IEEE 1284 EPP compliant communication interface. In other embodiments, other modes of communication could also be used (e.g., extended communication port (ECP) or serial communications) could be used without departing from the invention.

This process continues at 526 until all data are transferred at 530. The FPGA 220 then wakes up to operate with the new configuration parameters stored in configuration memory 424 at 534. The FPGA 220 continues to operate as configured at 538 until such time as the INIT line 416 is again asserted by the Host computer 210 at 544. Control then returns to 514 where the FPGA 220 is cleared and the reprogramming process proceeds as previously described.

Using this mechanism, the FPGA 220 can be coupled to the host computer 210 using a single interface 214 for both programming the FPGA 220 and for later communication with the FPGA 220 operating as the virtual microcontroller. This avoids use of multiple interface connections and/or use of a separate processor to handle details associated with configuration programming and communication with the FPGA 220.

The present invention provides for full in-circuit emulation without need for a special bond-out version of a DUT. This is accomplished using a minimal amount of design embedded within the DUT itself. In the current embodiment, the only functionality required of the production microcontroller itself is to provide for transfer of data over two lines forming the data portion of the interface and reading commands for break, watchdog and interrupt functions received over the same two data lines. These provisions are simple to implement, and use minimal circuitry. The two additional pinouts used for this function were readily accommodated in the eight bit microcontroller of the current invention. Moreover, the use of a single standard IEEE 1284 printer cable interface between the virtual microcontroller and the host computer to provide both FPGA programming and communication between the ICE system and the Host processor provides for a simple and versatile implementation.

As previously mentioned, microcontroller 232 is mounted on a pod assembly attached by a bus 226 to base station 218. Advantageously, in certain embodiments, the pod assembly can be used not only to carry microcontroller 232 for purposes of In-Circuit Emulation and debugging operations, but also to program another microcontroller. **FIGURE 8** illustrates a pod assembly 610 for carrying out such a programming process. In this embodiment, microcontroller 232 is mounted on the pod assembly and connected to the bus 226 as previously described. Additional connections are provided to microcontroller 232 for reset and power signals as previously described. In addition, a socket 620 is provided for carrying another microcontroller temporarily to provide programming functions for the microcontroller. All of the connections between the base station 218 and the pod can be made using a standard category 5 cable as previously described.

In order to carry out programming functions on a microcontroller mounted in socket 620, the operation of microcontroller 232 should be placed in a mode such that it does not disturb the programming process, i.e., a sleep mode. This is accomplished by placing microcontroller 232 in a sleep mode while simultaneously placing the microcontroller 620 in a programming mode. Programming data can then be clocked into the microcontroller in socket 620 from base station 218 without the operation being disrupted by the microcontroller 232.

In order to accomplish both debug operations and programming operations using pod assembly 610, microcontroller 232 can be selectively placed in either a debug or a sleep mode. **FIGURE 9** illustrates a process 700 starting at 702 for placing microcontroller 232 in a debug mode for running In-Circuit Emulation and debugging operations. This process starts at 702 after which power is turned on to the pod assembly 610 at 706. After power is turned on the In-Circuit Emulation system holds power and reset lines asserted (at a logic high) at 710 until the microcontroller's power stabilizes. After that, the in-circuit emulation system pulls the data0 line to a logic high level at 714 and then releases the reset line while the data0 signal is still at a logic high at 718.

This sequence of events tells the microcontroller 232 that it should enter the debug mode at 722. At this point normal In-Circuit Emulation operations and debug operations can be carried out and the microcontroller 232 begins executing its stored operational code.

FIGURE 10 depicts a process 800 used to place the microcontroller 232 in a sleep mode so that it does not disrupt programming operations of the microcontroller in socket 620. Of course, the microcontroller in socket 620 is placed in a programming mode while the microcontroller 232 is in a sleep mode in order to effect programming of the microcontroller in socket 620. Of course, when microcontroller 232 is in the debug mode, there is normally no microcontroller present in socket 620 to disrupt the debug operation.

Process 800 starts at 804 after which power is turned on to the pod assembly at 808. The In-Circuit Emulation system holds power and reset asserted at 812 and watches for the data0 line to go to a logic high indicating stable power at 816. At 820 the In-Circuit Emulation system pulls the data0 line to a logic low at 820 and then releases the reset line while holding the data0 line at a logic low at 824. This sequence of events causes the microcontroller 232 to enter the sleep mode.

The In-Circuit Emulation system then uses the data1 line as a programming clock line in order to clock data into the microcontroller in socket 620 at 824. Prior to clocking in programming data, a key code is clocked in at 828. This key code is specific to the particular microcontroller device being programmed in order to prevent unauthorized modifications of the program. If the key code does not match at 832, the microcontroller in socket 620 begins running whatever code is stored internally at 840. If the key matches at 832, the microcontroller in socket 620 enters a program mode so that it can accept program code. Data representing program code are then clocked into the programmable microcontroller in socket 620 at 848 by applying clock signals to the data1 line and data to the data0 line in order to program the socketed microcontroller being programmed. This process proceeds until all of the program lines have been clocked in at 852 after which the

microcontroller halts and power is turned off to the pod at 856. The programming process is now completed at 860 and the microcontroller in socket 620 can be removed for use.

Thus, an In-Circuit Emulation system pod assembly can be used not only to carry a real microcontroller 232 or other device under test but can double as a device programmer which is normally a separately purchased item that can costs the user several hundred dollars or more.

As described above, the microcontroller 232 is placed in a sleep mode when a device in socket 620 is to be programmed. This sleep mode is triggered by the release of reset while the Data0 line is held at a logic low. If reset is released with Data0 at a logic high, the microcontroller 232 boots into the debug mode. In the preferred embodiment, the reset pin on the device in socket 620 is hard wired to a logic low, so that when power is applied to the device in socket 620 and it receives the key code, the device enters the programming mode.

With the above-referenced Cypress Microsystems microcontroller family, the microcontroller looks for receipt of a valid key within about 16 milli-seconds from release of reset or valid power in order to enter the programming mode. Otherwise, whatever code is stored in the microcontroller's memory begins executing. Thus, the above sequence of events is carried out rapidly after power-up in order to cause the microcontroller in socket 620 to enter the program mode. To facilitate this and assure that the entire process is carried out within the time specified, the procedure is carried using a hardware state machine incorporated within the base station 218 and implemented as a part of the FPGA. However, this is not to be considered limiting.

While the present embodiment is implemented using a processor that does not use pipelined instructions, this is not to be considered limiting. As long as adequate time is available to serialize and transmit data over the interface, the present interface and break management techniques could equally well be implemented in a pipelined processor.

Those skilled in the art will understand that although the current invention has been explained in terms of providing in-circuit emulation of the core processing functions of a microcontroller. However, the present invention can be realized for any complex electronic device for which in-circuit emulation is needed including, but not limited to, microprocessors and other complex large scale integration devices without limitation. Moreover, although the mechanism for use of the interface between the host processor and the FPGA has been described in the environment of an ICE system, this should not be considered limiting since this interface mechanism can be used for other systems requiring FPGA programming and communication functions over a single interface.

Those skilled in the art will recognize that the present invention has been described in terms of exemplary embodiments based upon use of a programmed processor. However, the invention should not be so limited, since the present invention could be implemented using hardware component equivalents such as special purpose hardware and/or dedicated processors which are equivalents to the invention as described and claimed. Similarly, general purpose computers, microprocessor based computers, micro-controllers, optical computers, analog computers, dedicated processors and/or dedicated hard wired logic may be used to construct alternative equivalent embodiments of the present invention.

Those skilled in the art will appreciate that the program steps and associated data used to implement the embodiments described above can be implemented using disc storage as well as other forms of storage such as for example Read Only Memory (ROM) devices, Random Access Memory (RAM) devices; optical storage elements, magnetic storage elements, magneto-optical storage elements, flash memory, core memory and/or other equivalent storage technologies without departing from the present invention. Such alternative storage devices should be considered equivalents.

The present invention, as described in embodiments herein, is implemented using a programmed processor executing programming instructions that are broadly described above in flow chart form that can be stored on any suitable

electronic storage medium or transmitted over any suitable electronic communication medium. However, those skilled in the art will appreciate that the processes described above can be implemented in any number of variations and in many suitable programming languages without departing from the present invention. For example, the order of certain operations carried out can often be varied, additional operations can be added or operations can be deleted without departing from the invention. Error trapping can be added and/or enhanced and variations can be made in user interface and information presentation without departing from the present invention. Such variations are contemplated and considered equivalent.

While the invention has been described in conjunction with specific embodiments, it is evident that many alternatives, modifications, permutations and variations will become apparent to those skilled in the art in light of the foregoing description. Accordingly, it is intended that the present invention embrace all such alternatives, modifications and variations as fall within the scope of the appended claims.

What is claimed is: